**Edge Detection – Course Project Report**

EN605.617.FA – Introduction to GPU Programming

Professor Chancellor Pascale

Miles Gapcynski

12/10/2018

Table of Contents

[Table of Figures 2](#_Toc532122709)

[Project Overview 3](#_Toc532122710)

[Introduction 3](#_Toc532122711)

[Implementation 4](#_Toc532122712)

[Convolution and Kernels 4](#_Toc532122713)

[General Approach 5](#_Toc532122714)

[Results and Analysis 8](#_Toc532122715)

[Grayscale vs HSV 8](#_Toc532122716)

[Prewitt 8](#_Toc532122717)

[Roberts 9](#_Toc532122718)

[Sobel 10](#_Toc532122719)

[Timing Results 10](#_Toc532122720)

[Conclusion 12](#_Toc532122721)

[Lessons Learned 12](#_Toc532122722)

[Expansion Areas 12](#_Toc532122723)

[Works Cited 13](#_Toc532122724)

# Table of Figures

[Figure 1 - Convolution (Machine Learning Guru) 4](#_Toc532122725)

[Figure 2 - Roberts Operator (Crane, 86) 5](#_Toc532122726)

[Figure 3 - Prewitt Operator (Crane, 86) 5](#_Toc532122727)

[Figure 4 - Sobel Operator (Crane, 86) 5](#_Toc532122728)

[Figure 5 - Test Image and Converted Grayscale 6](#_Toc532122729)

[Figure 6 - Test Image and Hue 7](#_Toc532122730)

[Figure 7 - Saturation and Value 7](#_Toc532122731)

[Figure 8 - Prewitt Grayscale and Hue Output 8](#_Toc532122732)

[Figure 9 - Prewitt Saturation and Value Output 8](#_Toc532122733)

[Figure 10 - Roberts Grayscale and Hue Output 9](#_Toc532122734)

[Figure 11 - Roberts Saturation and Value Output 9](#_Toc532122735)

[Figure 12 - Sobel Grayscale and Hue Output 10](#_Toc532122736)

[Figure 13 - Sobel Saturation and Value Output 10](#_Toc532122737)

[Figure 14 - Timing Results Table 11](#_Toc532122738)

[Figure 15 - Timing Results Plot 11](#_Toc532122739)

# Project Overview

## Introduction

For my course project I decided to create a program that would compare how effective the CPU and GPU are when processing image data. I’ve always been interested in the topic of image processing, but I didn’t have much experience writing software that processes image data. I ended up choosing the topic of edge detection to focus my efforts and properly scope my project for the semester. What I didn’t truly grasp until now was that edge detection itself is actually a pretty deep topic, and it has a wide range of applications it can be used for. One example that Randy Crane provides on page 79 in his book Simplified Approach to Image Processing is that edge detection can be used by photo editing software for a magic wand tool, which draws a border around pixels that match a selected pixel’s value. Crane goes on to explain that edge detection can be used in other ways, such as for identifying objects or for image registration, which “aligns two images that may have been acquired at separate times or from different sensors” (79). The effectiveness of these different algorithms/techniques depends on many factors, such as how large the image is or if the algorithm’s intention is to find the vertical edges within a picture.

The edge detection algorithm I chose to implement for my course project is to convolve different 3x3 kernels with the image data in order to identify edges within a picture. The program reads in an image specified by the user, convolves the image’s pixel data with a 3x3 kernel, and outputs the convolution results as a new image. The program was created using C++ and Cuda so that it could be run on the CPU or the GPU, allowing me to compare how effective each device is at finding the edges within a given picture.

# Implementation

## Convolution and Kernels

I chose to use convolution for my edge detection algorithm because Randy Crane points out that “common gradient (or orthogonal gradient) operators” utilize convolution to “find horizontal and vertical edges” (86). In image processing, convolution involves multiplying each pixel and it’s neighbors with the corresponding positions in the kernel, and then performing a summation of the multiplication results. The kernel represents a mask that filters the image data, producing a desired result depending on the kernel’s values (weights). The kernel is generally applied to each pixel in an iterative fashion, sliding over each row and column one by one. There are several methods for handling neighbors outside of the image bounds, such as zero padding or wrapping, but I chose to ignore the outermost pixel rows/columns for simplicity. The following diagram helps illustrate how convolution of an image and kernel works:

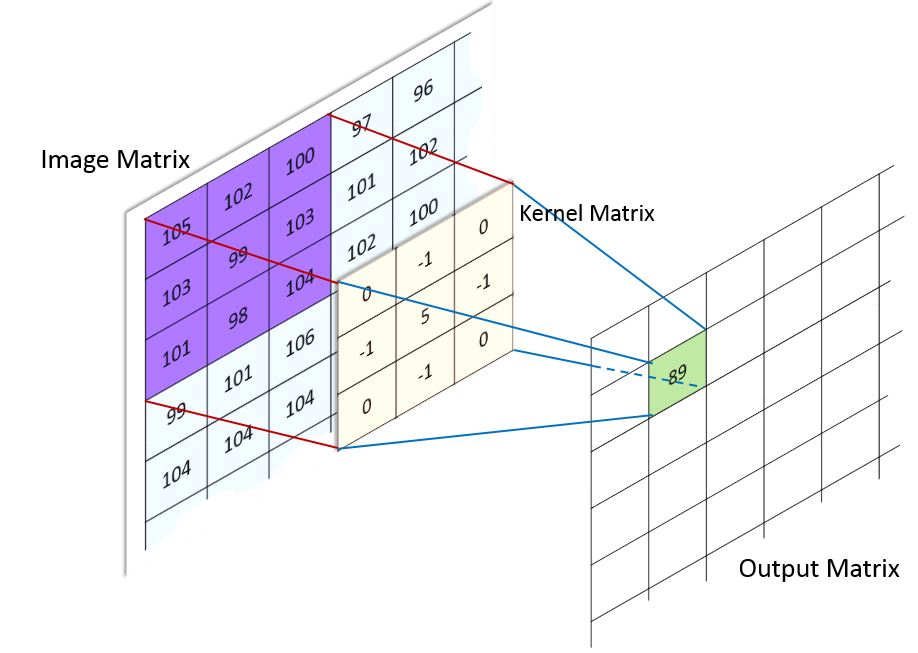


Figure 1 - Convolution (Machine Learning Guru)

As noted earlier, Crane mentioned that common gradient operators (kernels) find both the horizontal and vertical edges within a picture by convolving each pixel position and it’s neighbors with two different kernels (86). One kernel is designed to find edges in the horizontal direction, while another kernel is designed to find edges in the vertical direction. The edge detection algorithm then calculates the magnitude of the two convolutions to produce a value representing the intensity of the edge at each pixel location. I chose three gradient operators to compare the results of: Roberts, Prewitt, and Sobel.

The Roberts operator, illustrated in figure 2, “has a smaller effective area than other masks, making it more susceptible to noise. The other masks are better able to average out fluctuations” (Crane, 86).

Figure 2 - Roberts Operator (Crane, 86)

The Prewitt operator, illustrated in figure 3, “is more sensitive to vertical and horizontal edges than diagonal edges” (Crane, 86).

Figure 3 - Prewitt Operator (Crane, 86)

Finally, the Sobel operator, illustrated in figure 4, “is more sensitive to diagonal edges than vertical and horizontal edges” (Crane, 86).

Figure 4 - Sobel Operator (Crane, 86)

## General Approach

I started the project by determining the best way to read and store image data from a file. The structure of different file formats can vary quite a bit, but the image data generally consists of a set of pixels that can have 1 to N channels of values per pixel. The channel values can be represented by different bit depths, and the channel values for each pixel can be stored in either an interleaved or sequential format. Due to the varied nature of image file formats, I decided it would be best to utilize a library that specializes in reading/writing image files. I chose to utilize FreeImage, which is an open-source library that we used earlier in the semester to handle reading/writing image files. I wrote some utility methods that interfaced with FreeImage in order to map the image data into channel-separated RGB values. This means that all of the red channel values were stored in the array first, then the green values, and then the blue values last. I chose to perform this mapping was because I found it easier to index into the array to retrieve the channel values for each pixel and it’s neighbors.

Once I was able to read different image formats with different numbers of channels, I then focused on how to convolve the image data and a kernel. For grayscale images I can simply iterate over the rows and columns of the image data, convolving the row and column kernels with each 3x3 section of pixels. This same approach can be done for each channel in a color image. However, since I didn’t need to retain the color information to find the edges within an image, I decided to convert the RGB values to grayscale before convolving the image data. In order to convert to grayscale, I created another utility function that averages the RGB channel values of each pixel. The following pictures show one of the images I used for testing, and that same image converted to grayscale:



Figure 5 - Test Image and Converted Grayscale

Once I had my strategy down for convolving grayscale and color images, I then implemented the convolution algorithm on the CPU. This was pretty straight forward, as it simply required iterating over every pixel of the image and applying the gradient operator (row and column kernels) to each 3x3 set of pixels. As stated earlier, I chose to ignore the outermost pixel rows/columns to avoid dealing with neighbors outside of the image boundaries. After that, the magnitude of the row and column convolution results needs to be calculated to find the intensity of the edge at that pixel location.

When it came time to implement the convolution algorithm on the GPU I knew I needed to reuse as much code as possible between the CPU and GPU in order to properly compare the two devices. This turned out to be rather straight forward, as the code that applied a filter to image data on the CPU was already processing a single pixel at a time in an iterative fashion. Because of this, I was able to move the logic that operated on a single pixel into a method that was used on the host and device. This meant that the only difference between the CPU and GPU was having to copy the filter and image data to the device when executing with the GPU. As for executing the GPU kernel, I chose to utilize a two-dimensional grid made up of 16 x 16 blocks (256 threads) to process the image data. I also created two different kernels, one using all global memory, and one using shared memory for the gradient operators, to see if shared memory would improve performance on the GPU.

One last thing to note about the project was that while researching the topic of edge detection I came across many instances of people pointing out that color image data can be represented by more than just the RGB color model. One model that kept popping up in discussions was the hue, saturation, and value (HSV) model. Hue is an angular value on a color wheel that indicates how similar a color is to red, green, or blue. Saturation refers to the intensity or purity of a color, and is found using chroma information. Value represents the maximum value of the RGB components that makeup a pixel. I decided to implement the RGB to HSV conversion alongside my RGB to grayscale functionality in order to compare how the three gradient operators I used would perform on different color models. The following pictures show the same bird image I used for testing, and that image converted to HSV:



Figure 6 - Test Image and Hue



Figure 7 - Saturation and Value

# Results and Analysis

I’ve broken my findings down into two sections: How the gradient operators compare when operating on grayscale data versus hue, saturation, and value (HSV) data, and how the timing information for how long the CPU and GPU took to find the edges within a picture.

## Grayscale vs HSV

This section showcases how the gradient operators performed when operating on grayscale data versus HSV data.

### Prewitt

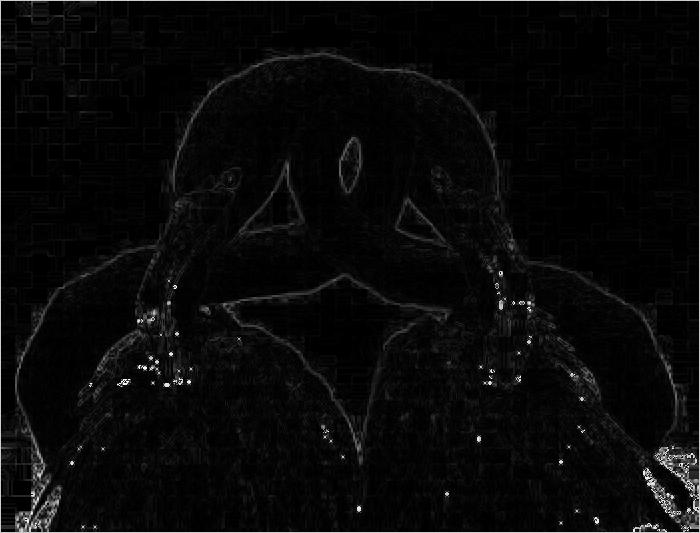
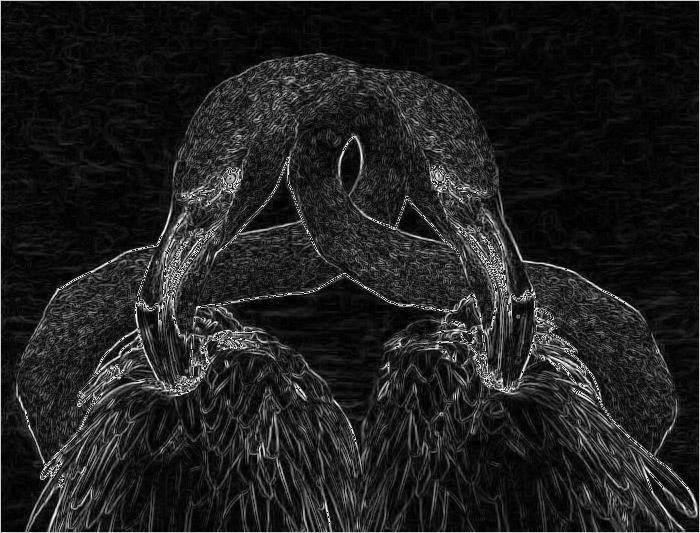


Figure 8 - Prewitt Grayscale and Hue Output

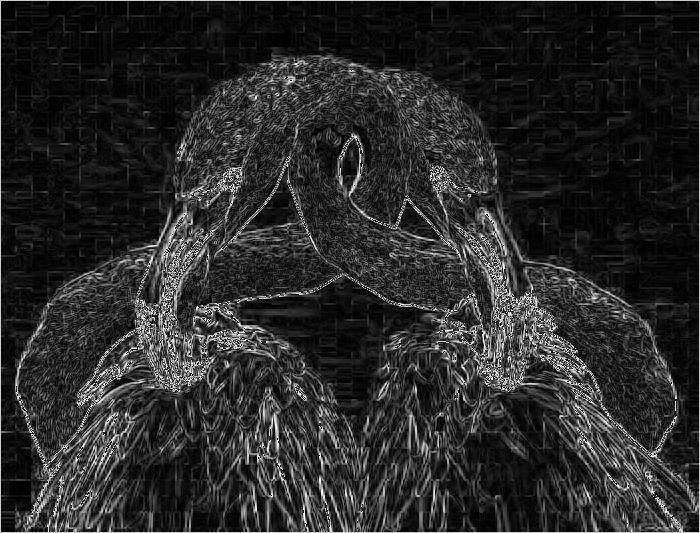


Figure 9 - Prewitt Saturation and Value Output

Upon examination, it seems that grayscale and value output could be useful when looking for less distinct edges within the picture, such as the edges around the feathers. Contrast that with the hue output, which seems like it could be more useful in an object recognition application as it highlighted the birds themselves. Also, the saturation highlights the bird’s eyes and beak, which seems like it could be useful in something like a facial recognition application. However, the saturation output may need to be cross-referenced with some of the other output data in order to filter out some of the noise. As noted earlier, the Prewitt operator seems to be more sensitive to vertical and horizontal edges, as opposed to diagonal ones (Crane, 86).

### Roberts

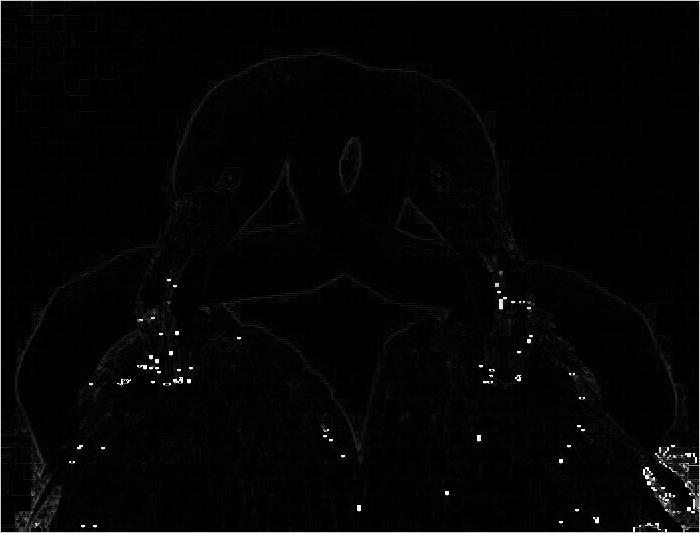
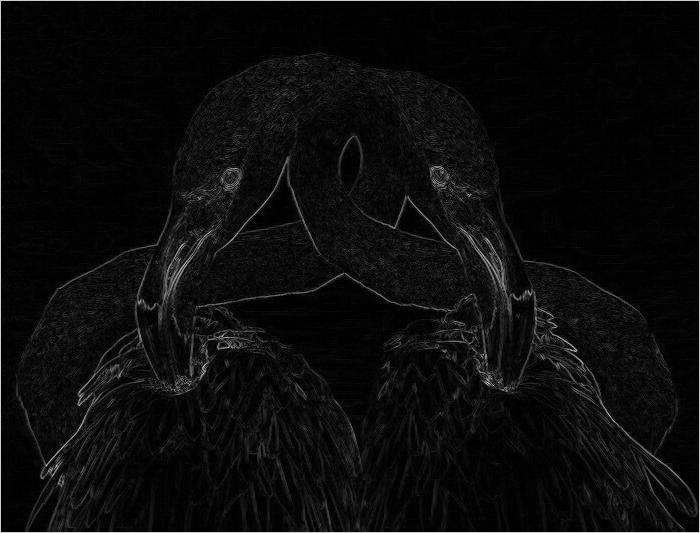


Figure 10 - Roberts Grayscale and Hue Output

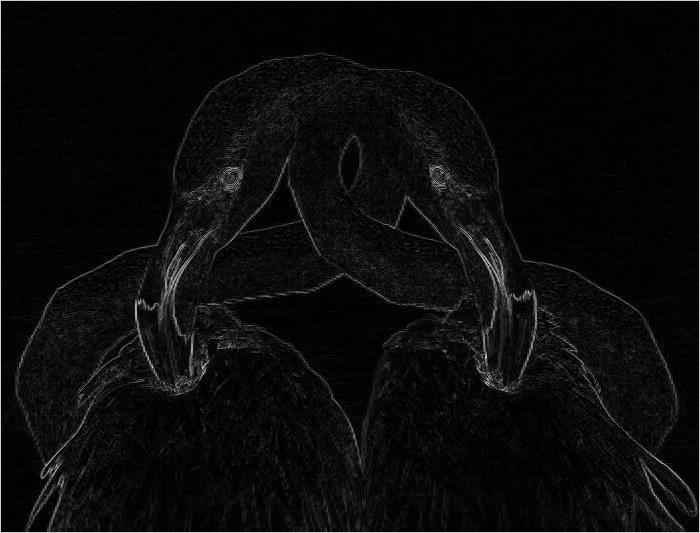
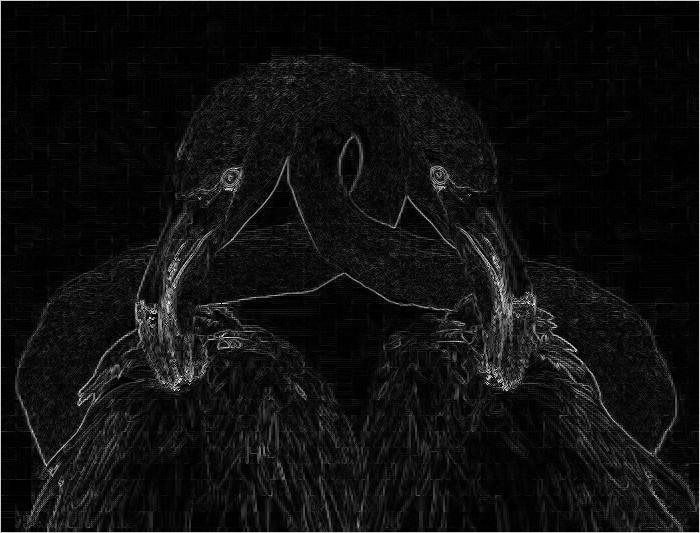


Figure 11 - Roberts Saturation and Value Output

Upon examination, the Roberts operator seems to do a better job of highlighting the birds in the image when compared to the Prewitt operator. The output also seems to be less noisy than the Prewitt operator output, which seems to contradict Crane’s findings, though this could just be due to the fact that the image isn’t noisy enough to see a degradation in the Roberts operator results. Beyond that, the grayscale versus HSV comparisons found in the Prewitt operator section seem to matchup with the Roberts operator output.

### Sobel

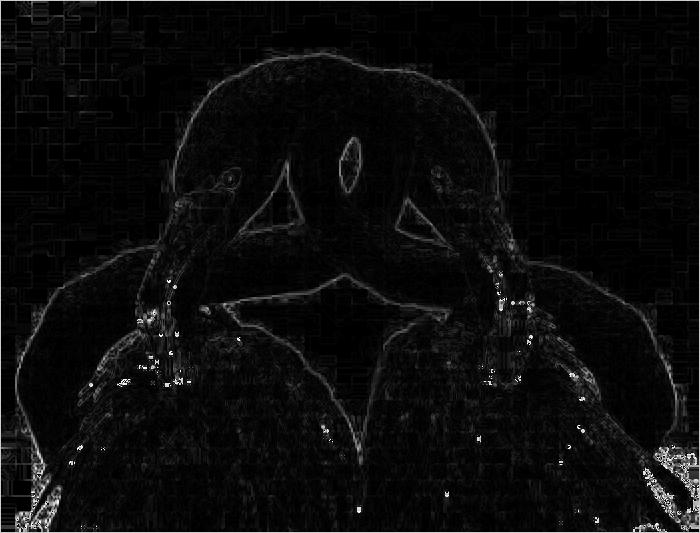
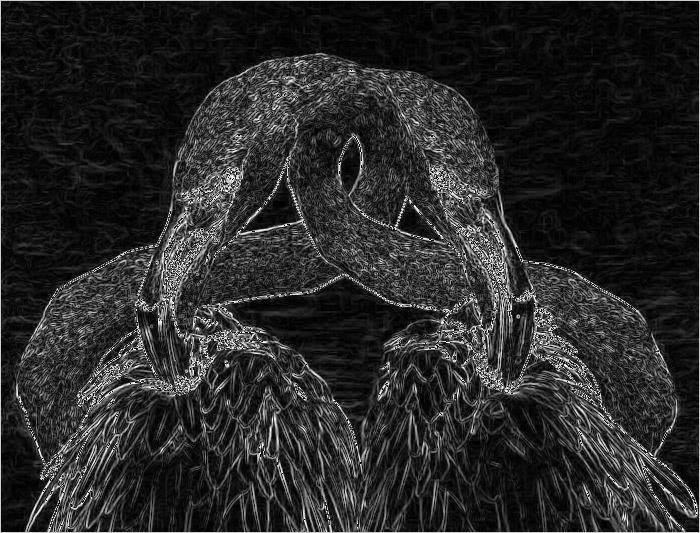


Figure 12 - Sobel Grayscale and Hue Output

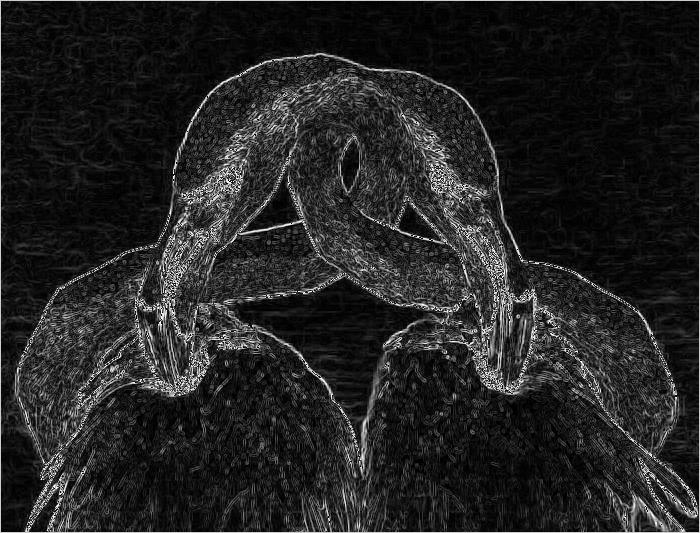
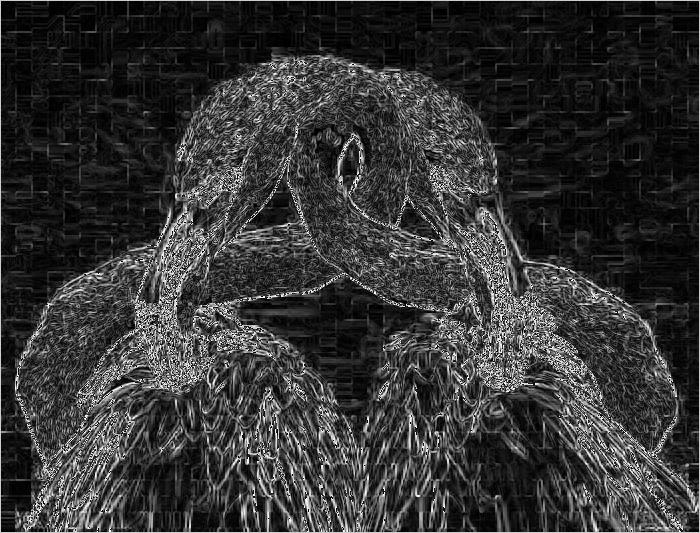


Figure 13 - Sobel Saturation and Value Output

Upon examination, it seems that Crane’s findings hold true for the Sobel operator, wherein the operator seems to handle diagonal edges better than the Prewitt operator. However, the Sobel operator output appears to have more noise when compared to the Prewitt operator output. In addition, the grayscale versus HSV findings in the Prewitt operator section still hold true here, though I would most likely opt for one of the other two gradient operators if I were writing an object or facial recognition application.

## Timing Results

The following table and line plot showcase the amount of time the CPU and GPU took to filter image data of varying sizes with a gradient operator. I did not list the different operators or color models in the table, as the math is the same regardless of which operator or color model is used. The results between the CPU and GPU are roughly what I expected, in that the GPU performed considerably better than the CPU at filtering the image data, especially as the image data size grew. The only surprising result was that the shared memory kernel did not perform as well as the global memory. Processing each pixel requires accessing the filter memory multiple times, which I thought would counteract the amount of time needed to copy the filter data to shared memory and sync the threads in each block. The shared memory setup time for the filter data apparently took longer than simply accessing the filter data in global memory.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Image | Image Dimensions (total pixels) | CPU Execution Times (ms) | GPU Execution Times (ms) | |
| Global Memory | Shared Memory |
| mountains.jpg | 500 x 348 (174,000) | 10.6232 | 0.5930 | 0.6353 |
| lena.png | 512 x 512 (262,144) | 16.0494 | 0.7689 | 0.9739 |
| birds.jpg | 700 x 533 (373,100) | 24.1710 | 1.1357 | 1.1654 |
| church.jpg | 1024 x 768 (786,432) | 48.7879 | 1.9197 | 2.2372 |
| pisa.jpg | 1600 x 900 (1,440,000) | 88.7504 | 3.5310 | 3.7038 |
| valley.jpg | 2560 x 1440 (3,686,400) | 228.5247 | 7.8206 | 8.3551 |
| town.jpg | 3840 x 2160 (8,294,400) | 516.8103 | 17.3303 | 18.7000 |

Figure 14 - Timing Results Table

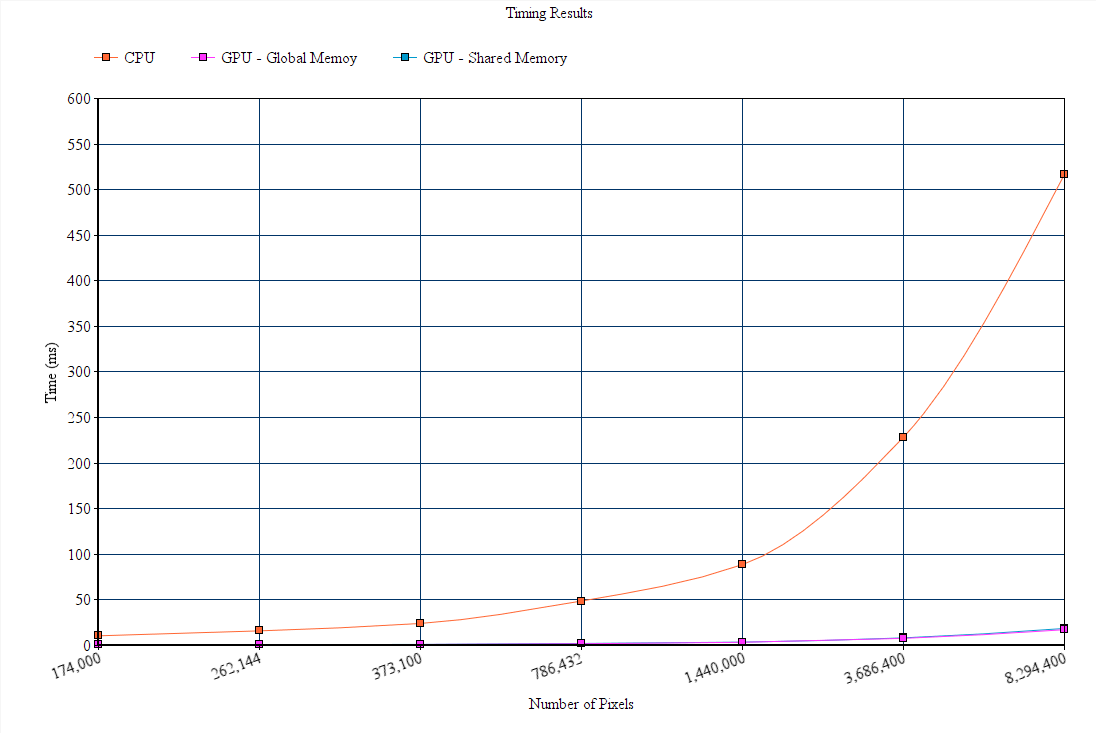


Figure 15 - Timing Results Plot

# Conclusion

## Lessons Learned

* Issues ran into along the way

## Expansion Areas

* It could be extended by comparing the results to other edge detection algorithms implemented in the same fashion (implemented on the CPU and GPU).
* Another extension of this project could involve passing the resulting image to an image recognition algorithm that searches for specific objects or patterns.
* Using texture memory
* Using a library like NPP
* Pixel thresholds to eliminate noise
* Different block size (16 x 16) for better occupancy?
* Compare cudaMalloc vs cudaAllocHost?

# Works Cited

Crane, Randy. Simplified Approach to Image Processing

Machine Learning Guru http://machinelearninguru.com/computer\_vision/basics/convolution/image\_convolution\_1.html